Lasagna Master

Welcome to Lasagna Master on Exercism's Go Track. If you need help running the tests or submitting your code, check out `HELP.md`. If you get stuck on the exercise, check out `HINTS.md`, but try and solve it without using those first :)

Introduction

A function allows you to group code into a reusable unit. It consists of the `func` keyword, the name of the function, and a comma-separated list of zero or more parameters and types in round brackets.

Function Parameters

All parameters must be explicitly typed; there is no type inference for parameters. There are no default values for parameters so all function parameters are required.

```
import "fmt"

// No parameters
func PrintHello() {
    fmt.Println("Hello")
}

// Two parameters
func PrintGreetingName(greeting string, name string) {
  fmt.Println(greeting + " " + name)
}
```

Parameters of the same type can be declared together, followed by a single type declaration.

```
import "fmt"

func PrintGreetingName(greeting, name string) {
  fmt.Println(greeting + " " + name)
}
```

Parameters vs. Arguments

Let's quickly cover two terms that are often confused together: `parameters` and `arguments`. Function parameters are the names defined in the function's signature, such as `greeting` and `name` in the function `PrintGreetingName` above. Function arguments are the concrete values passed to the function parameters when we invoke the function. For instance, in the example below, `"Hello"` and `"Katrina"` are the arguments passed to the `greeting` and `name` parameters:

```
PrintGreetingName("Hello", "Katrina")
```

Return Values

The function parameters are followed by zero or more return values which must also be explicitly typed. Single return values are left bare, multiple return values are wrapped in parenthesis. Values are returned to the calling code from functions using the `return` keyword. There can be multiple `return` statements in a function. The execution of the function ends as soon as it hits one of those `return` statements. If multiple values are to be returned from a function, they are comma separated.

```
func Hello(name string) string {
  return "Hello " + name
}

func HelloAndGoodbye(name string) (string, string) {
  return "Hello " + name, "Goodbye " + name
}
```

Invoking Functions

Invoking a function is done by specifying the function name and passing arguments for each of the function's parameters in parenthesis.

```
import "fmt"

// No parameters, no return value
```

```
func PrintHello() {
    fmt.Println("Hello")
}
// Called like this:
PrintHello()

// One parameter, one return value
func Hello(name string) string {
  return "Hello " + name
}
// Called like this:
greeting := Hello("Dave")

// Multiple parameters, multiple return values
func SumAndMultiply(a, b int) (int, int) {
    return a+b, a*b
}
// Called like this:
aplusb, atimesb := SumAndMultiply(a, b)
```

Named Return Values and Naked Return

As well as parameters, return values can optionally be named. If named return values are used, a `return` statement without arguments will return those values. This is known as a 'naked' return.

```
func SumAndMultiplyThenMinus(a, b, c int) (sum, mult int) {
    sum, mult = a+b, a*b
    sum -= c
    mult -= c
    return
}
```

Pass by Value vs. Pass by Reference

It is also important to clarify the concept of passing by value and passing by reference.

First, let's clarify passing by value. In the example below, when we pass the variable `val` to the function `MultiplyByTwo`, we passed a copy of `val`. Because of this, `newVal` has the updated value `4` but the original variable `val` is still `2`. Behind the scene, Go essentially makes a copy of the original value so that only this copy, a.k.a. `v`, is modified by the function.

```
val := 2
func MultiplyByTwo(v int) int {
    v = v * 2
    return v
}
newVal := MultiplyByTwo(val)
// newval is 4, val is still 2 because only a copy of its value was passed into the funct
```

Strictly speaking, all arguments are passed by value in Go, i.e. a copy is made of the value or data provided to the function. But if you don't want to make a copy of the data that is passed to a function and want to change the data in the function, then you should use `pointers` as arguments, a.k.a. pass by reference.

Pointers

We use a `pointer` to achieve passing by reference. By passing `pointer` arguments into a function, we could modify the underlying data passed into the function instead of only operating on a copy of the data.

For now, it is sufficient to know that pointer types can be recognized by the `*` in front of the type in the function signature.

```
func HandlePointers(x, y *int) {
    // Some logic to handle integer values referenced by pointers x and y
}
```

If the concept of `pointer` is confusing, no worries. We have a dedicated section later in the syllabus to help you master pointers.

Exceptions

Note that `slices` and `maps` are exceptions to the above-mentioned rule. When we pass a `slice` or a `map` as arguments into a function, they are treated as pointer types even though there is no explicit `*` in the type. This means that if we pass a slice or map into a function and modify its underlying data, the changes will be reflected on the original slice or map.

Instructions

In this exercise you are going to write some more code related to preparing and cooking your brilliant lasagna from your favorite cookbook.

You have four tasks. The first one is related to the cooking itself, the other three are about the perfect preparation.

1. Estimate the preparation time

For the next lasagna that you will prepare, you want to make sure you have enough time reserved so you can enjoy the cooking. You already planned which layers your lasagna will have. Now you want to estimate how long the preparation will take based on that.

Implement a function `PreparationTime` that accepts a slice of layers as a `[]string` and the average preparation time per layer in minutes as an `int`. The function should return the estimate for the total preparation time based on the number of layers as an `int`. Go has no default values for functions. If the average preparation time is passed as `0` (the default initial value for an `int`), then the default value of `2` should be used.

```
layers := []string{"sauce", "noodles", "sauce", "meat", "mozzarella", "noodles"}
PreparationTime(layers, 3)
// => 18
PreparationTime(layers, 0)
// => 12
```

2. Compute the amounts of noodles and sauce needed

Besides reserving the time, you also want to make sure you have enough sauce and noodles to cook the lasagna of your dreams. For each noodle layer in your lasagna, you will need 50 grams of noodles. For each sauce layer in your lasagna, you will need 0.2 liters of sauce.

Define the function `Quantities` that takes a slice of layers as parameter as a `[]string`. The function will then determine the quantity of noodles and sauce needed to make your meal. The result should be returned as two values of `noodles` as an `int` and `sauce` as a `float64`.

```
Quantities([]string{"sauce", "noodles", "sauce", "meat", "mozzarella", "noodles"})
// => 100, 0.4
```

3. Add the secret ingredient

A while ago you visited a friend and ate lasagna there. It was amazing and had something special to it. The friend sent you the list of ingredients and told you the last item on the list is the "secret ingredient" that made the meal so special. Now you want to add that secret ingredient to your recipe as well.

Write a function `AddSecretIngredient` that accepts two slices of ingredients of type `[]string` as parameters. The first parameter is the list your friend sent you, the second is the ingredient list of your own recipe. The last element in your ingredient list is always `"?"` . The function should replace it with the last item from your friends list. **Note:** `AddSecretIngredient` does not return anything - you should modify the list of your ingredients directly. The list with your friend's ingredients should **not** be modified. Also, since `slice` is passed into a function as pointers, changes to the two `[]string` arguments passed into `AddSecretIngredient` will be modified directly.

```
friendsList := []string{"noodles", "sauce", "mozzarella", "kampot pepper"}
myList := []string{"noodles", "meat", "sauce", "mozzarella","?"}

AddSecretIngredient(friendsList, myList)
// myList => []string{"noodles", "meat", "sauce", "mozzarella", "kampot pepper"}
```

4. Scale the recipe

The amounts listed in your cookbook only yield enough lasagna for two portions. Since you want to cook for more people next time, you want to calculate the amounts for different numbers of portions.

Implement a function `ScaleRecipe` that takes two parameters.

- A slice of `float64` amounts needed for 2 portions.
- The number of portions you want to cook.

The function should return a slice of `float64` of the amounts needed for the desired number of portions. You want to keep the original recipe though. This means the `quantities` argument should not be modified in this function.

```
quantities := []float64{ 1.2, 3.6, 10.5 }
scaledQuantities := ScaleRecipe(quantities, 4)
// => []float64{ 2.4, 7.2, 21 }
```

Source

Created by

- @bobtfish

Contributed to by

- @sougat818